

**F/6 9/2**

F49620-77-C-0099

AFOSR-TR-80-1010

NL

$$\frac{1}{2} \left( \frac{1}{2} + \frac{1}{2} \right) = \frac{1}{2}$$

END  
DATE  
FILMED  
12-80  
DTIC

12-80

AD A091385

LEVEL

4077

896

5

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY

DDG FILE COPY

NOV 3 1963

C

MCDONNELL DOUGLAS

CORPORATION

Approved for public release;  
distribution unlimited.

80 10 21 03

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER <b>AFOSR-TR- 80 - 1010</b>	2. GOVT ACCESSION NO. <b>AD-A091385</b>	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  <b>METRICS OF SOFTWARE QUALITY</b>		5. TYPE OF REPORT & PERIOD COVERED  <b>Final</b>
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)  <b>Z. Jelinski and P. B. Moranda</b>		8. CONTRACT OR GRANT NUMBER(s)  <b>F49620-77-C-0099</b>
9. PERFORMING ORGANIZATION NAME AND ADDRESS <b>McDonnell Douglas Astronautics Company 5301 Bolsa Avenue Huntington Beach, California 92647</b>		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS  <b>61102F 2304/A2</b>
11. CONTROLLING OFFICE NAME AND ADDRESS <b>Air Force Office of Scientific Research <i>N/M</i> Bolling Air Force Base District of Columbia 20332</b>		12. REPORT DATE <b>August 1980</b>
		13. NUMBER OF PAGES <b>53</b>
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) <b>Unclassified</b>
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  <b>Approved for public release; distribution unlimited</b>		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  <b>Software metrics Test tools</b>		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) <b>This report covers the period from 1 June 1979 to 31 May 1980 and concerns the third of three phases. This phase concentrated on the use of the augmented Program Testing Translator described in the July 1979 Interim Report on AFOSR F44620-74-C-008.</b>  <b>The software required to test FORTRAN programs, first with random numbers and then by preselected constructed cases, was developed during this interval. Software modifications were made to permit performance measures</b>		

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (Continued)

of the degree of coverage to include the results due to use of constructed cases. The automation of the mix of random numbers and constructed cases was studied and limitations were found.

Two new models for use during the development phase of programming were developed. These models apply to programs which grow in size during the debugging period.

UNCLASSIFIED

MCDONNELL  
DOUGLAS  
CORPORATION

**METRICS OF SOFTWARE QUALITY.**

AUGUST 1980

MDC-G8584

ANNUAL REPORT, 1 JUN 1979 - 31 MAY 1980

CONTRACT

F49620-77-C-0099

PREPARED FOR:  
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH  
BOLLING AIR FORCE BASE, DISTRICT OF COLUMBIA 20332

ATTENTION: LT. COL. GEORGE W. MCKEMIE, CONTRACT MONITOR  
DIRECTORATE OF MATHEMATICAL AND INFORMATION SCIENCES

**MCDONNELL DOUGLAS ASTRONAUTICS COMPANY-HUNTINGTON BEACH**

5301 Bolsa Avenue Huntington Beach, California 92647 (714) 896-3311

**DISTRIBUTION STATEMENT A**

Approved for public release;  
Distribution Unlimited

389210

## SUMMARY

This report covers the period from 1 June 1979 to 31 May 1980, and concerns the third of three phases. This phase concentrated on the development and application of the augmented Program Testing Translator (PTT) described in the July 1979 Interim Report on AFOSR F44620-74-C-008.

The software required to test FORTRAN programs, first with random numbers and then by preselected constructed cases, was developed during this interval. Software modifications were made to permit performance measurements of the degree of coverage to include the results due to the use of constructed cases. The automation of the mix of random numbers and constructed cases was studied and limitations were found.

Two new models for use during the development phase of programming were developed. These models apply to programs which grow in size during the debugging period.

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)  
NOTICE OF TRANSMITTAL TO DDC

This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b). Distribution is unlimited.

A. D. BLOSE  
Technical Information Officer

## Section 1

### OBJECTIVES AND TASK DESCRIPTIONS

#### 1.1 WORK ACCOMPLISHED

The primary effort during the contract period encompassed work on three tasks:

1. Tailor or expand the testing programs that were developed in the early phases of the contract.
2. Code so as to provide valuations of the program predicates, and values of the artificial program variables which provide the data for the search procedures.
3. Modify, install, and test the tool on a laboratory computer when the scope and size of the test tool are established.

#### 1.2 ADDITIONAL WORK REQUIRED

Continuing studies will be made to assess the practicality of a fully automated version of testing.

This additional work is included in the major task:

Test the tool and the methodology, using the several constructs (connection matrix, status vectors, predicate valuations, and input and output data) through the implementation on a "laboratory" type of computer, such as the Nanodata QM-1.

Section 2  
STATUS OF RESEARCH EFFORTS

2.1 SEGMENT COVERAGE BY RANDOM AND CONSTRUCTED CASES

2.1.1 Problem Identification

This particular part of the study reached the point where the methods of analysis were established and were tested out acceptably. There are three major software related problems, which were solved. They are discussed in the following three subsections.

2.1.1.1 Problem 1: Estimation of Number of Residual Tracks

First, the estimated number of tracks through a program obtained by using random numbers as program drivers was solved. This problem had been solved in principle, but implementation of it heretofore had been effected by the tedious process of desk checking segment usages against all past usages. The software required to automate the process has been written.

Although the capability exists to use arbitrary probability distributions for the random drivers, the usual procedure was to employ the same distribution and the same range of values for all input variables, this common distribution was a uniform distribution on a fixed range (of the logarithm). For additional flexibility, two additions to the total APTT program were added. First, the range of each real input variable was made selectable, with the distribution over the logarithm uniform on this range; second, for integers, the selection was made from a uniform distribution over whatever range may be selected. A random (50/50) sign selection was also made for the (always positive) real variables; for the integer variables the range may be extended to negative integers so that the device is not required.

The selection of random values for the input variables (real or integer) provides the set of values for one run. The procedure employed for estimating the number of tracks that will be exercised requires a number of executions and comparisons. In the automatic version, the track that accompanies one input data (random) selection is identified in terms of a zero or one assignment to the arbitrarily ordered set of segments which comprise the list of



segments: a zero for nonusage and a 1 for one or more usages. (Two paths which differ in their nonzero counts of the usages of segments, or in their order of execution, are considered to have the same track).

In the implementation of the estimation process, the above outlined initial portion is followed (in the postprocessor) by a routine which compares the sequence of binary n-tuples (one "ordinate" for each program segment) in order to accomplish two things:

A. Establish whether a newly examined track is the same as some track earlier examined, effected by comparing the n-tuples ordinate by ordinate against all previously taken tracks,

B. Marking the trial number of the current track by a zero or 1 in accordance with the results of the comparisons, a zero if an "old" n-tuple has been found and a 1 if the examined track is new.

The data for the estimation procedure consist of the pattern of 0's and 1's obtained in the above comparisons. The primary observable consists of the total trials between adjacent 1's. These spacings between 1's are reported as  $X_1, X_2, \dots, X_n$  and represent the difference in the indices representing trial numbers:  $X_1$  is the separation between the first trial number (by definition, the first trial results in the first new track) and the trial number which produces the second new track (usually this separation is 1 because of the high likelihood that a new data set will produce a different track);  $X_2$  is the separation between the third and second new track, etc.

With data  $X_1, X_2, \dots, X_n$  obtained by running the program over T trials, the number of new tracks is estimated from the equation

$$\sum_{i=1}^n \frac{1}{N-(i-1)} = \frac{nT}{NT - \sum_{i=1}^n (i-1)X_i} \quad (1)$$

where N is the unknown,  $X_i$  are as defined, T is the total number of trials and n is the number of  $X_i$  employed.

The augmented version of APPT achieves this entire process of comparison and estimation automatically.

#### 2.1.1.2 Problem 2: Comprehensive Coverage

A certain track is taken in response to any input data set. This track is characterized by the segments that are exercised, without regard to their order or their multiplicity of execution. The major problems in completely automating the cover-testing process are in construction of the software required to establish the status of testing, maintain suspense files on all unexercised program segments, insert augmenting variables corresponding to predicates which define the entry into the (unexercised) computational segments, search the input variable space to achieve entry, compare the resulting track with previously obtained tracks, and prune the original tracks to a set of smaller dimension (manifested in the reduction of the original n-tuples to tuples of smaller size).

The complete list of tasks required to develop the software follows:

- A. Identify unexercised branches (at the end of the initial runs with random numbers).
- B. Pick an unexercised branch and display the listing associated with the branch (a "back" sort is required which identifies the instruction number of the involved predicate).
- C. Formation of an auxiliary variable based on the nature of the predicate. (For example, if the test,  $A < B$ , is the predicate, the auxiliary variable could be  $C_1 = B - A$ ).
- D. Create a variable (with requisite modifications to the object program).
- E. Vary input variables until the auxiliary variable is positive. Rationale for the variation depends on the program variables identified in the listing.
- F. List all exercised segments and compare with preceding usage.
- G. "Release" the variable and proceed to a new unexercised branch.
- H. In an extension of the above procedure, several auxiliary variables can be inserted at one time and input data chosen in some systematic way (a search) to achieve arbitrary valuations on all auxiliary variables.

Not all of the software required for this problem has been written and, in the main, the process of integrating a display into a human/computer/display interrogative mode has not been accomplished. The progress which has been made toward that end is described in Section 2.1.2.

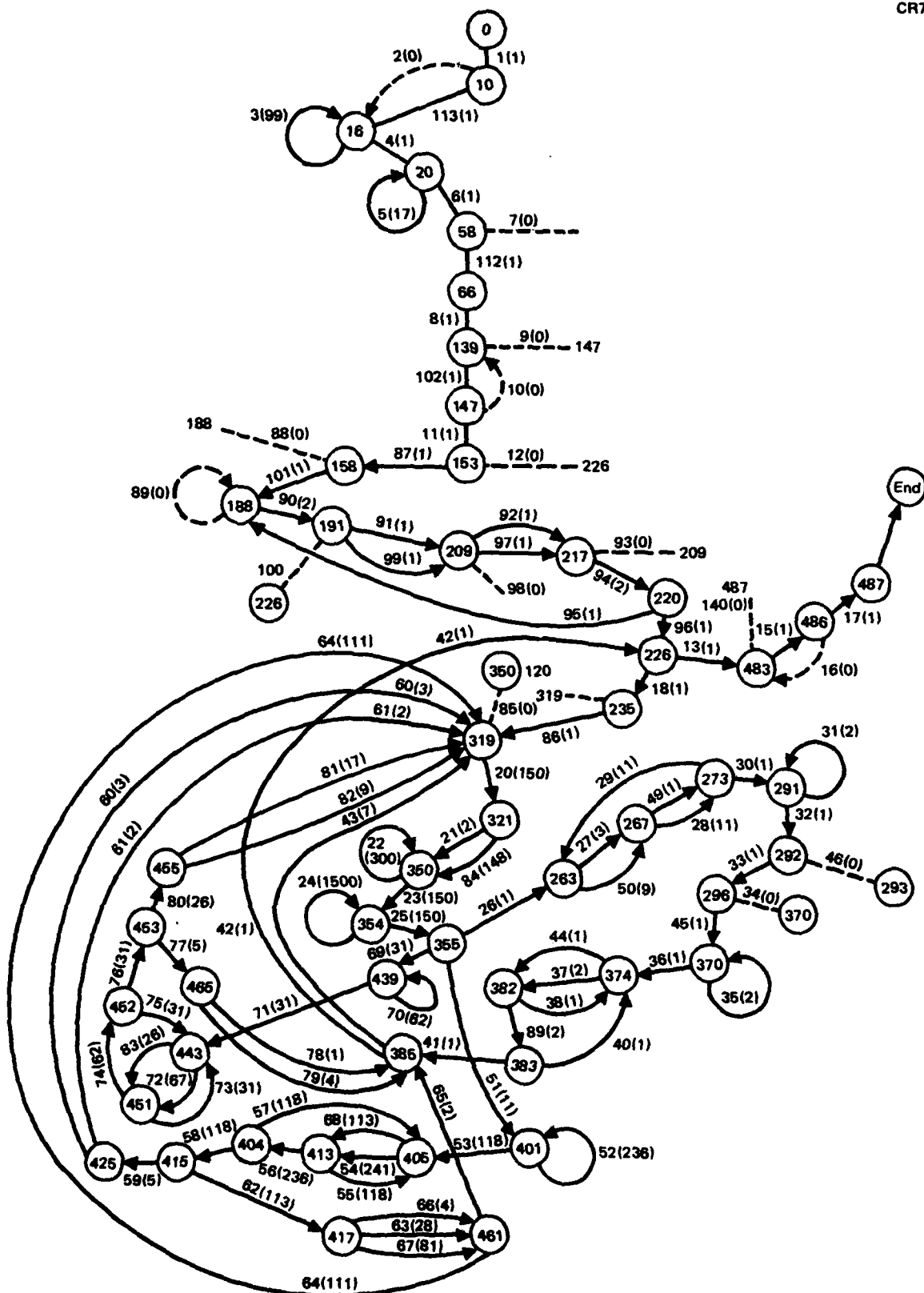
#### 2.1.1.3 Problem 3: Formation of Execution Sequences

It is well to state at the outset that only the outline of this problem has been established. The following paragraphs describe the background and outline of the problem.

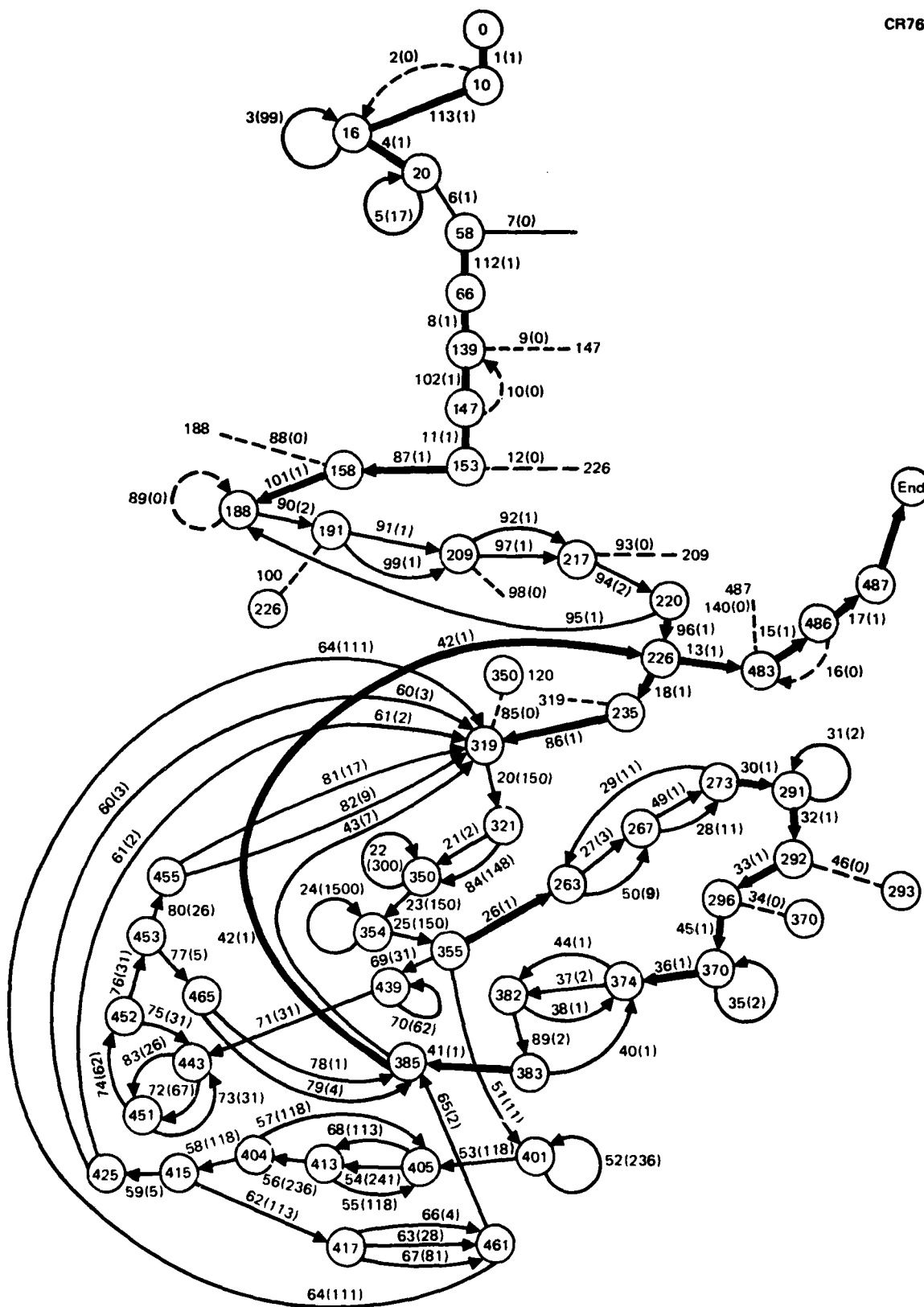
The use of tracks as proxies for execution sequences is in part necessary and in part expedient. Tracks are necessary because one usually cannot determine the actual sequence from a list of usages: with several entrances and several exits from a node and a different usage number of each, there is usually no way to determine the actual sequence of the computation that would produce the usage numbers. On the other hand, information often is available which would allow the program flow to be determined in a gross or general way, and that information heretofore has not been employed in our studies. It would be helpful to program testers to provide a general sequence of the flow resulting from a given input driver set.

To illustrate this, an example, depicted in Figure 1, shows the set of executed segments and their counts as solid lines or arcs between all nodes which were passed during the first data set employed on the ORLA program described in the 2nd Interim Report (Reference 1). It will be noted that dotted lines are also shown emanating from certain of the nodes which were passed. These are branches which were not taken on the run; they would be important in coverage testing but can be ignored for the present discussion. The flow of the computations can be determined unambiguously only in the cases where a single execution is performed on a segment and no other segment parallels the segment. For example, there is such a segment joining nodes 355 and 263 in the central lower one-third of the chart. This and others are highlighted in Figure 2, where they may be easier to locate.

The general flow can be formed from the unambiguous segments which show a usage of one. In one case, there are (at node 226) two segments, both with a count of 1, shown exiting a node. But this particular ambiguous case



### Figure 1. Response to First Data



### Figure 2. General Flow of Computations

is easily resolvable (i. e., precedence determined) because the branch along segment 13, joining nodes 226 and 483, joins to the exit (END), and so cannot precede the segment joining nodes 226 and 235. This suggests an interesting problem of which the preceding example is the most trivial: given a set of nodes and their counts, determine under what conditions the actual flow can be determined. This "academic" problem will not be pursued in this study.

The application of the simple rule which establishes the one-time used segments (a "footprint," or better, a "one-print") permits a linking of certain segments to form contiguous blocks of the program, the General Flow of the title of Figure 2.

Such linkings are shown in Figure 2, where the defined flow consists of the following:

Block 1: Segments 1, 113, 4, 6, 112, 8, 102, 11, 87, 101

Block 2: Segments 96, 18, 86

Block 3: Segment 26

Block 4: Segments 30, 32, 33, 45, 36

Block 5: Segments 41, 42, 13, 15, 17 (END)

Even the undefined flow can be combined to form pseudo segments if there are no dotted lines: thus, the series/parallel segments 20, 21, 84, 22, 23, 24, and 25, which are between nodes 319 and 355, can be treated as a single pseudo segment with a usage of 150, the entry and exit counts at the two joined nodes. In addition to these pseudo segments, another type of merging is possible in certain areas. For example, some of the segments from Block 4 of the above list can be joined with the segment of Block 3 to form a superblock. Since all possible paths to and from nodes 263 and 273 have been exercised, these can be eliminated from further consideration, permitting formation of a pseudo segment with which to join segment 30 to segment 26. Also, since node 291 has all exits exercised, it too can join to form a larger block (26, pseudo segment, 30, and 32). Because node 292 has a dotted line out of it, there is no further merging possible between the two blocks.

Even though the remainder of the program flow is undefined, there are many points which are internal to the undefined blocks where reduction is

possible. A trivial example is the pair of parallel paths 91 and 99 between nodes 191 and 209, which can be merged into a two-use segment; more interesting cases can be identified in the lower left portion of Figure 2. Thus, between nodes 401 and 415 are segments 53, 68, 54, 55, 57, 56, and 58, all of which can be merged to a 118-use pseudo segment.

Figure 3 shows a considerably pruned version of the flow diagram. As with the preceding, it is developed from the one-prints and more is required to establish the sequence. For example, segment 42 appears to follow (dynamically) 41, but there is no reason to think a priori or in a local context that it actually does. In a global context, however, it is known that segment 42 is the later exit from node 385, because 42 joins to 226 and from there out to the END.

## 2.1.2 Software Implementations

### 2.1.2.1 The Program Testing Translator

The PTT and its augmentation, APTT, are the primary software tools supporting the Software Quality Metrics effort. Use of the PTT tool in earlier phases of the study allowed individual subroutine driving by employing random variables for each subroutine parameter. In applications, data was used to drive the subroutine repeatedly; this was followed by collecting statistics on execution counts and branchings. Then a postprocessing step was performed which presented, in tabular form, a list of segments and the corresponding segment usage statistics. This process in the early phases required a manual connecting of segments into possibly larger segments. Noticeably absent from the early version of the PTT tool were branches implied in DO-LOOP termination, and in END= and ERR= branches in file operations.

To alleviate some of the shortcomings, the PTT was modified in several respects. First, it was altered to operate on the entire program instead of subroutines. This allowed analysis of all FORTRAN modules in a program to occur at one time. The PTT was also upgraded to recognize the implicit DO, END, and ERR branches. Along with these changes, the segment analysis was completely redesigned to allow composition of complete dynamic

1



segments automatically and, to a small degree, to pinpoint unreachable portions of code. This redesign also caused a more efficient allocation of segment monitors.

#### 2.1.2.2 Augmentations of PTT

Since the redesign, the three problems discussed in Section 2.1.1 have arisen and these required further modifications to the PTT. The first was to obtain, automatically, an estimate of the number of residual tracks remaining to be found. The solution involved addition of an input description record by the user to control ranges on the random variables used as drivers, and to get a new set of input variables upon calling of the input routine. The gathered execution statistics numbered each case run and determined, with use of the postprocessor, the  $X_i$  required to calculate the residual number of tracks.

The second problem required software assistance in the selection of test data, first by identifying the unexercised branches and, second, to allow the user to add one or more new temporary auxiliary variables to be monitored. Cases would then be run and the values of the input variables varied until a positive valuation of the temporary auxiliary variables occurred. The choice of the auxiliary variables and the equations which define them are left to the user in this version of PTT. The PTT assists the user by compiling so as to incorporate the auxiliary variables, generating a monitoring code, and by updating the list of unexercised branches.

The third problem, discussed in Section 2.1.1.3, was to assist the user with a method of determining the program flow. This problem presented a level of difficulty such that no immediate assistance from the PTT per se was deemed possible.

#### 2.1.2.3 Examples of Applications

To illustrate each of the first two problems and the PTT solution, the matrix triangularization example will be reexamined. This example is extensively discussed in the 2nd Annual Report (Reference 1), where directed graphs of the potential program flow and examples of the coverage by random numbers and constructed cases were given. Listings of the MAIN and TRIANGULARIZATION subroutine comprise Figure 4.

```

0- 1      PROGRAM AMM
      IMPLICIT INTEGER(A-Z)
      INTEGER IP(3)
      COMMON /C(4,3),X
      OPEN(UNIT=20,DEVICE='DISK:',FILE='TRIANG.RES',ACCESS='SEQUENT
      I=3
      N=4
      J=3
      DO 10 K=1,N
          CALL INPUT(A)
          WRITE(20,101)K,((A(I,J),J=1,N),I=1,N)
          CALL TRIANG(IP,A,N)
          WRITE(20,102)K,((A(I,J),J=1,N),I=1,N)
      10- 11 CONTINUE
      12 STOP
      13 FORMAT(' BEFORE TRIANGULARIZATION ('//I1,'')//3(3X,F20.10))
      14 FORMAT(' AFTER TRIANGULARIZATION ('//I1,'')//3(3X,F20.10))
      END
      CSOINPUT
      VARIABLES=1,
      ROUTINE=INPUT,
      A(4,3)=9999(-2,1);

```

Figure 4. Listing of an Example Program (Page 1 of 2)

```

SUBROUTINE TRIANG(VF,A,B)
  IMPLICIT INTEGER(A-Z)

  LOGICAL IP(3)
  REAL A(4,3),T

  IP(N)=1
  DO 6 K=1,N
    VF(K,EQ,N) GOTO 5
    KPI=K+1
    N=K
    DO 1 I=KPI,N
      IF(ABS(A(I,K)).GT.ABS(A(N,K))) M=1
    CONTINUE
    IP(N)=N
    IF(A(N,N) IP(N)=-IP(N)
    T=A(N,N)
    A(N,K)=A(K,N)
    A(K,K)=T
    IF(T.EQ.0) GOTO 5
    DO 2 I=KPI,N
      A(I,K)=-A(I,K)/T
    DO 4 J=KPI,N
      T=A(N,J)
      A(N,J)=A(K,J)
      A(K,J)=T
    IF(T.EQ.0) GOTO 4
    DO 3 I=KPI,N
      A(I,J)=A(I,J)+A(N,K)*T
    CONTINUE
    VF(A(K,K).EQ.0) IP(N)=0
  CONTINUE
  RETURN
  END

```

Figure 4. Listing of an Example Program (Page 2 of 2)

Appendix A contains tables and reports of the PTT output for three separate test runs. The reports show the testing coverage provided by using the user-described input routine INROUT. INROUT returns a new set of random distributed data values for the input matrix A. The data values are uniformly distributed over the logarithm in the range -2 to 1. The sign of the individual data items is also selected randomly.

The first three cases of test run No. 1 (see Pages A-1 to A-3 in Appendix A) show coverage of code for the MAIN program as 100% in the column marked Summary. Subroutine TRIANG gets a summary coverage of 86.96%. The remaining segments to be tested are numbers 3, 12, and 16, as seen in the segment reference report (Page A-2 of Appendix A). The segment reference tables are used to relate the segment numbers and their corresponding program statement numbers together. As an example, it is seen that segment 3 contains lines 34, 35, 36, and 37 in subroutine TRIANG (see Figure 4). These lines correspond to:

```

6      IF(A(K,K).EQ.0)34 IP(N) = 035
      CONTINUE → K = K+136 IF (K.LE.N)37 loop
(DO-loop termination includes an implied conditional branch)

```


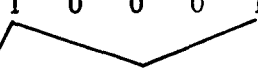
Following the summary reports and the segment reference tables, the trial statistics appear on Page A-3, for example. These are the  $X_i$  that are needed to calculate the estimate of the number of remaining tracks. (Actually, more than three cases are required for the estimation and the three entries on Page A-3 form only a part of the data used.)

Supplied as part of the testing package is a program that interacts with the user and calculates the difference of the two sides of equation (1) in Paragraph 2.1.1.1 based on trial solutions supplied by the user.

To explain how the  $X_i$  are formed the formation of  $X_1$  and  $X_2$  will be considered. Case 1 of run 1 (see Page A-1) shows the number of times each segment of MAIN and TRIANG were executed. Since this is the first test case, the first unique track is automatically formed. Case 2 of run 1 for TRIANG (Page A-1) shows the same segments being executed (the number of

executions of each segment listed happen to be the same, but this is irrelevant; the comparison is made on the basis of whether or not the segment was executed, not on how many times) as in case 1, run 1. The MAIN routine shows a difference in execution. Therefore case 1 and case 2 are different, so we form  $X_1 = 1$ . This means that one case occurred since the last unique track. If we compare case 3 of run 1 against cases 1 and 2 we also find a difference in the MAIN routine (see segment 3 execution counts). This gives us our third unique track. Hence,  $X_2 = 1$ , also, since only 1 case occurred since the last unique track.

Continuing in this fashion, by comparing cases 1 through 9 (in Appendix A), in order, we find unique tracks for cases 1, 2, 3, 4, and 8. (A summary of the nine cases is found on Page A-9.) The Boolean tokens associated with the sequence are shown below:

CASE	1	2	3	4	5	6	7	8	9
NEW/ OLD (=1,=0)	1	1	1	1	0	0	0	1	0
									
	$X_1=1 \ X_2=1 \ X_3=1$				$X_4=4$				

By using the estimation equation, it was determined that there existed 9.1 new tracks to be found.

Appendix B contains reference tables of the PTT output for a constructed case. The constructed case shows the use of monitor variables (Page B-2). For constructed cases, the user is required to supply input data to the program, and to supply the monitor variables. It is seen that the user-supplied input is in the DATA statement in the MAIN program. Subroutine TRIANG shows the use of monitors inserted into the program of branch points.

By analyzing the unexercised segments, 3, 12, and 16 of the three test runs of Appendix A, where they are marked by asterisks in all three of the segment reference tables (Pages A-2, A-5, A-8), it can be determined

from the listing that the variable T holds the key to exercising these segments. Further examination suggests that if A[3,3] is equal to zero then segment 3 will be exercised.

Segment 12 requires variable T to be zero. For this to be true, A[1,1] could be equal to zero or A[3,2] could be greater than A[2,2] and A[1,2] must be equal to zero.

Segment 16 also requires variable T to be equal to zero. This condition will result if A[1,1] is not zero and A[1,2] is equal to zero.

These findings determined the initial values of A for the DATA statement. By observing the segment reference for subroutine TRIANG, we find that segments 3, 12, and 16 have been executed and the test coverage is complete.

## 2.2 ERROR-DETECTION MODELS

### 2.2.1 Summary

Two variations of the Jelinski-Moranda model were developed for estimation during program development. The first permits estimation of the error content of the completed software package using data which is taken on only portions of the package. That model is applicable when the eventual size of the program is known at the outset.

The second model permits a similar analysis during the development of any software package which is homogeneous with respect to its complexity (error making and finding).

These models should assist analysts in an early determination of error content. They should also eliminate the present practice of applying models to the wrong regime (decreasing failure rate models applied to growing-in-size software).

### 2.2.2 Introduction

In normal usage of the Jelinski-Moranda model, the software package under test is assumed to be of fixed size with a fixed number of incipient

errors. The size of the package does not appear explicitly in the model as a parameter, and its effect is only indirectly realized by the way it affects the number of incipient errors which exist at the start of testing (there is a direct relation between instruction count and error count).

That model could not be employed legitimately on software packages which were incomplete. Several workers attempted to fit the model to an initial period of time when its error rate was, indeed, increasing, due to the growing size, and they met with no success. (As a matter of fact, the only models which produced reasonable estimates when applied during this regime, were the increasing failure rate models).

It would be helpful if, at the outset, an estimate could be obtained of the total error count which will be realized in test and usage of a package.

Recent work by IBM (Reference 2) has prompted a reexamination of the original Jelinski-Moranda model for the purpose of incorporating the (changing) program size. This turns out to be very easily effected if good record keeping can be maintained during program development so that the size of the package is recorded as a function of some convenient timing metric (CPU or calendar). Following is a description of the analysis.

The original model is depicted in Figure 5, where the two parameters are shown in Figure 5(b), and a typical realization of the error-finding process is shown in Figure 5(a).  $N$  is the initial error content (of a completed program) and  $\phi$  is the contribution to the error rate due to a single error.

While the meaning of  $\phi$  is maintained in the two models, the meaning of "initial error content" needs to be clarified. This is done below in the description of Model 1, where, in effect,  $N$  maintains its meaning as the number of errors in a completed package. In the second model, a fixed error rate per instruction is assumed, and growth of the package is measured by the count of instructions (under test) versus time.

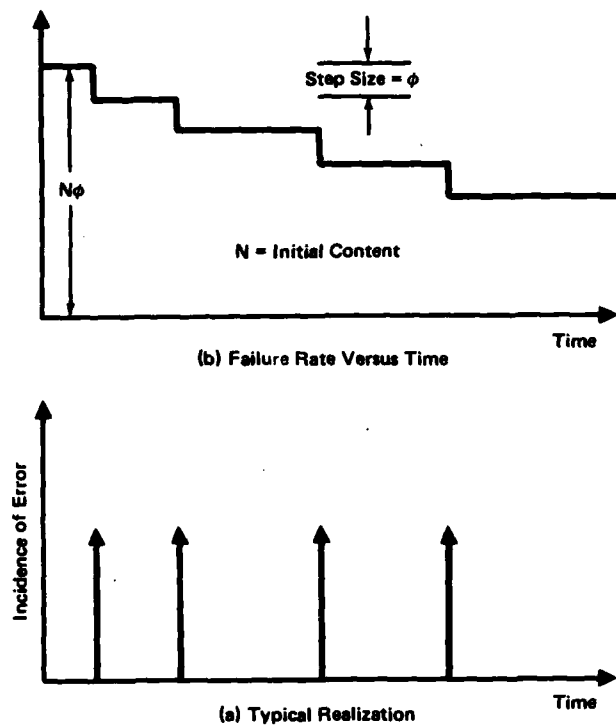


Figure 5. Purification Process and its Realization

#### 2.2.2.1 Model 1

Let  $S(t)$  denote the fraction of the total number of statements which a complete program will have. The metric  $t$  is measured in terms either of the accumulated CPU time, or of the amount of calendar time, which has been used for testing the package.

The simplest way of introducing the effect is to use  $S(t)$  as a "modulation" of the error detection rate  $Z(t)$  of the original model. In the notation formerly employed, this combined or modulated rate, denoted  $W(t)$ , is:

$$\begin{aligned}
 W(t) &= S(t) Z(t) \\
 &= S(t) [(N-i+1)\phi] \text{ for } T'_{i-1} \leq t \leq T'_i
 \end{aligned} \tag{2}$$

and  $T'_1, T'_2, T'_3, \dots$ , denote the times of detection for the errors. (Primes are employed on  $T$ 's to distinguish them from the times of the original process.) The effect of  $S(t)$  on the  $T'_i$  should be made clear at the outset. When



$S(t)$ , the fraction of the total count, increases, the composite error rate will generally increase, as will the liability for error for the "modulated" process. For this reason, the times  $T'_i$  for the composite process,  $W(t)$ , will be different from the  $T_i$  of the  $Z(t)$  process. Since  $N$  in the original model represented the total error content of a complete software package, a proper correspondence which preserves the meaning is that  $N$  is the error content at a time corresponding to the completion of the software package,  $S(t) = 1.00$ . This necessarily presumes that the size of the package which will be developed is known at the outset. ( $S_0(t)$  would represent the fraction of the total which is accomplished at time  $t$ .) This may or may not be a serious barrier. Some modules can be sized at the outset, but large complex programs may not be. An alternative to this is offered subsequently in Model 2.

For the present model,  $S_0(t)$  is a nondecreasing function which starts at zero at  $T'_0$  and achieves its maximum value at some unknown-at-the-outset time,  $T'_c$ .

Thus,  $0 \leq S_0(t) \leq 1$ , with  $S_0(T'_0) = 0$  and  $S_0(T'_c) = 1$ .

While  $S_0(t)$  is, in the large sense, random, the records of progress will permit specific values of  $S_0(t)$  to be determined and the randomness is of no concern. In particular, it is necessary that  $S_0(t)$  can be determined at the epoch times  $T'_1, T'_2, \dots, T'_n$  at which the errors are detected.

When the completion time,  $T'_c$ , is reached and for times thereafter, the software package is complete ( $S_0(T'_c) = 1$ ) and, formally, the density given in Equation (1) is the same as that given in the original paper (Reference 3).

It has been mentioned earlier that the time pattern of errors will be different for the "modulated" process, and it is interesting to see just what would happen if  $S_0(T'_0)$ , or for short,  $S_0(0)$ , were 0.10 (10% of the package is initially available for test), and it did not increase beyond that for a long period of testing. The time pattern of errors  $T'_1, T'_2, \dots, T'_n$  which would occur, would have associated separations  $X'_1 = T'_1 - T'_0$ ,  $X'_2 = T'_2 - T'_1$ , ...,  $X'_n = T'_n - T'_{n-1}$ .

Because  $S_0(0) = 0.10$ , the composite detection rate for the first error would be  $(0.10) N\phi$ , that is, 10% of the original error-detection rate. This means that the first detection time  $T_1^1$ , would (on the average) be 10 times as long as the time for the corresponding error of the unmodulated process. The second error would have the same property (on the average), and so forth. The implications of this fact can be seen from the following. The likelihood function would be

$$L(X_1^1, X_2^1, \dots, X_n^1) =$$

$$\prod_{i=1}^n S_0(0)\phi[N-(i-1)] \exp \{ -[S_0(0)\phi(N-i+1)X_i^1] \}.$$

The likelihood equations obtained by differentiating the logarithm of the likelihood with respect to  $N$  and  $\phi$  are:

$$\left. \sum_{i=1}^n \frac{1}{N-(i-1)} - S_0(0) \phi \sum_{i=1}^n X_i^1 = 0 \right\} \quad (3a)$$

and

$$\left. \frac{n}{\phi} - S_0(0) \sum_{i=1}^n [N-(i-1)]X_i^1 = 0 \right\} \quad (3b)$$

As noted above, the observables  $X_i^1$  would be (about or on average) 10 times as large as before. Thus, from Equation (3b), the solution  $\hat{\phi}$  will be (on average) the same as its value for the unmodulated process, or for the completed software package.

Using the solved-for value of  $\hat{\phi}$  in Equation (3a) and the fact that  $S_0(0)X_1^1$  in the new process is the same as  $X_1$  in the original process, it is seen that the solution  $N$  is also the same as before.

The analysis then shows that if it is known that a package under test represents (in all respects) a certain percentage of the total, then the total eventual error content can be estimated by using these slightly modified likelihood equations.

The result is encouraging for the outlook for success in the following simple generalization of the above example. In this generalization, the  $S_0(t)$  modulating function is constrained to be constant during each test interval. Using essentially the same notation as before, the likelihood equations for the generalized modulated process are

$$\sum_{i=1}^n \frac{1}{N-i+1} - \phi \sum S_{i-1} X_i' = 0 \quad (4a)$$

and

$$\frac{n}{\phi} - \sum_{i=1}^n S_{i-1} (N-i+1) X_i' = 0 \quad (4b)$$

where  $S_{i-1}$  is the percentage completion achieved prior to the start of the  $i$ th interval.

Solutions for the parameters can be carried out as indicated above in the example.

The mean-time-to-next error MTTF ( $n+1$  st in the present context) can be estimated by evaluating the rate at time  $T_n'$  and taking the reciprocal of it. In the present case (using a subscript on the left side to correspond to the model number):

$$MTTF_1 = \frac{1}{S(T_n') \hat{N} - n) \phi},$$

where  $\hat{N}$  and  $\hat{\phi}$  are solutions to the Maximum Likelihood Equations (MLE's).

#### 2.2.2.2 Model 2

Let  $E_p$  denote a characteristic rate of error-making for the programmer (or programmer team) and the program type. This rate will be estimated by application of the model described subsequently, but there are some useful facts concerning this parameter.

In 1975, it was observed (Reference 4) that there appears to be a "... 'universal' coding - error rate....," which has a value of about 2 errors per 100 instructions (of the language in which the program has been written). This observation was based primarily on the data (now famous) provided by F. Akiyama, but also on earlier observations made by B. J. Hatter, et. al. Subsequently, the validity of this "thermodynamically stable" parameter has been reinforced by several other studies.

The interesting feature of some of this later data (Reference 5) is that the error rate of two per hundred was observed on programs which had completed their development and integration phases; they were under test before the relevant error counting was initiated. This is surprising since the coding error rate is thought of as being similar to a typist's miskeying, and should be purifiable by edit routines and by code checking due to early mis-starts of the program.

These features of an hypothesized entity are fortunately not used in the following analysis.

The error rate at any point in the development of a program whose current instruction count is  $G(t)$  is assumed to be proportional to the current error content

$$V(t) = \phi [G(t) \cdot E_p - n(t)] \quad (5)$$

where  $n(t)$  is the accumulated number of error corrections, and  $E_p$  is the per instruction error rate.

As before, if  $G(t)$  can only change at error-discovery epochs,  $T_1, T_2, \dots, T_n$ , and, if  $n(t)$  also has this feature, then the rate has the form

$$V(t) = \phi [G_{i-1} E_p - (i-1)] \text{ for } T_{i-1} \leq t \leq T_i \quad (6)$$

where  $G_{i-1} = G(T_{i-1})$ , and  $n(t)$  is  $i-1$  for the interval starting at  $T_{i-1}$ .

Since  $G(t)$  is a function or process which takes place without any apparent dependence on the error-finding process (except that the error epochs are assumed to be the points of entry of new code) it is reasonable to assume that the random time separations between errors ( $X_1, X_2, \dots, X_n$ ) are statistically independent.

Under these conditions, the constant rate implies an exponential distribution for the  $X_i$ , and the likelihood function for  $n$  errors is:

$$L(X_1, X_2, \dots, X_n) = \prod_{i=1}^n \phi [G_{i-1} E_p - (i-1)] \exp \{ -\phi X_i [G_{i-1} E_p - (i-1)] \} \quad (7)$$

The MLE's obtained by differentiating the logarithm of the likelihood function with respect to  $\phi$  and  $E_p$  are:

$$\sum_{i=1}^n \frac{G_{i-1}}{G_{i-1} E_p - (i-1)} - \phi \sum_{i=1}^n G_{i-1} X_i = 0 \quad (8a)$$

$$\frac{n}{\phi} - \sum [G_{i-1} E_p - (i-1)] X_i = 0 \quad (8b)$$

The MLE's are solved as before: Equation (8b) can be algebraically solved for  $\phi$ ; this is substituted in Equation (8a), and the resulting key equation is solved for  $E_p$  by trial and error.

It is recalled that the desired performance parameter is  $E_p$ , which can then be used with either the current (known) or projected (estimated) instruction count to determine the total error content.

Estimates of the MTTF at any time can be obtained by the formula

$$MTTF_2 = \frac{1}{\hat{\phi}[G_n \hat{E}_p - n]} \quad (9)$$

### 2.2.3 Conclusions

The two models presented in the analysis are both very tractible analytically.

Model 1 would be of use for those programs whose eventual size is known at the outset. It requires that a record of the times of error occurrences be maintained as well as a record of the percentage of completion at each of the error-detection times. It provides, at any stage of testing, an estimate of the error content of the untested complete package.

Model 2 applies to any developing software package which is homogenous with respect to the complexity of programming and with respect to the talents of the programmers. The important property is that  $E_p$ , the error-making rate (or error-finding rate), must be a constant across the entire software package. In case of inhomogeneity separate analyses are advised.

### 2.2.4 Glossary

Terms and symbols used in the preceding sections are identified as follows:

$S_o(t)$	A "modulation function" which ranges from $0 \leq S_o(t) \leq 1.0$ and is nondecreasing. It represents the fraction of the code completed up to time $t$ . It is a given for the problem.
$Z(t)$	The Jelinski-Moranda detection or purification process.
$W(t)$	The product of $S(t)$ and $Z(t)$ . It represents the error-making or error-detecting rate versus time for Model 1.

$N$	The number of errors in the completely coded software package. This is estimated from data.
$\phi$	The contribution of one error to the detection (failure) rate. This is estimated from data.
$T'_1$	The time at which the $i^{\text{th}}$ error is found, measured in any convenient timing metric. An observable.
$X'_1$	The separation between the $i^{\text{th}}$ and the $i-1^{\text{st}}$ error. An observable.
$n$	The cumulative number of errors found in testing up to time $T'_n$ .
$S_i$	Is the percentage of completion during the $(i+1)^{\text{st}}$ interval. This is provided as exogenous data.
$MT\hat{T}F_i$	The estimated meantime to error obtained by using Model $i$ ( $i = 1, 2$ ).
$G(t)$	The nondecreasing function representing the total instruction count of the package at time $t$ . This is a given for the problem.
$E_p$	The error making rate for a given program-programmer mix. It is estimated from data.
$n(t)$	The number of errors found during test up to time $t$ . An observable.
$V(t)$	A stochastic process representing error-making or error-detecting rate versus time.
$L(X_1, X_2, \dots, X_n)$	The generic representation for the likelihood function.

### Section 3 PUBLICATIONS

#### 3.1 STATUS OF PREVIOUSLY REPORTED PUBLICATIONS

1. P. B. Moranda, "Event-Altered Rate Models for General Reliability Analysis," published in the IEEE Transactions on Reliability (December 1979).

2. P. B. Moranda, "Probability-Based Models for the Failures During Burn-in Phase," under review for publication in the Journal of the Operations Research Society of America.

3. P. B. Moranda, "Comments on: Competing Software Reliability Models," to appear in IEE Transactions on Software Engineering, September 1980.

4. P. B. Moranda, "Comments on: How to Measure Software Reliability and How Not To," incorporated in a referee report to the author on material submitted to a workshop on quantitative software models, Kiamesha Lake, New York, 9-11 October 1979.

#### 3.2 NEW WORK IN PROGRESS OR PUBLISHED

P. B. Moranda, "Error Detection Models for Application During Program Development," submitted to IEEE Transactions on Reliability, under revision for resubmittal.

#### 3.3 COMMENTS AND REVIEWS

P. B. Moranda, Review of "Statistical Methods in Computer Performance Analysis," in Current Trends in Programming Methodology, for Computing Reviews, February 1980.



Section 4  
KEY PERSONNEL

ZYGMUNT JELINSKI - Branch Chief, Computer Sciences  
BS, Economics and Statistics, University of London, 1952.  
MBA, Business Administration, Metropolitan College, England, 1953.  
MS, Mathematical Statistics, University of London, 1954.

Mr. Jelinski has been managing computer research and development programs for 23 years. As Branch Chief, Computer Sciences, at MDAC-HB, he currently directs research in software reliability, modeling, validation and verification, language processing, emulation, and simulation. Software tools developed and/or maintained under his direction include the Program Evaluator and Tester (PET), SUMC Meta-assembler, Compiler Writing System, OPAL Development Tool (OPALDET), CAMIL, and the Generalized Language Processor (METRAN). He was study manager of a contract to develop methodology for effective test case selection (a research contract for the National Bureau of Standards), and he directed a study for NASA on methodology for reliable software. Another program under his direction was one in which software validation methodology was developed and applied to a tactical software system, resulting in accurate prediction of software malfunctions. He also directed the Software Reliability Study sponsored by the Air Force Office of Scientific Research. This study involved research into the development and evaluation of mathematical models representing the pattern of software malfunctions. At Rockwell International, Mr. Jelinski was Chief of Systems Programming Technology. He directed design and retrieval systems and engineering design aids. Earlier, he managed all programming for the RECOMP II and III computers. At Philco Corporation, he was manager of programming in support of Philco 2000 computer marketing, and at RCA he was manager of applications for RCA 4310 communication computers.

Mr. Jelinski's publications include the following:

HOLDET - Higher Order Language Development and Evaluation Tool (coauthor), MDAC Paper WD 2769, presented to Computers in Aerospace Conference, Los Angeles, November 1977 (AIAA, NASA, IEEE, ACM).

An Approach to Solution of Problems with Support Software as Deliverables, MDAC Paper WD 2759, presented to Defense Systems Management Review, Ft. Belvoir, Virginia, March 1978.

Recent Software Development Techniques in the United States, MDAC Paper WD 2706, presented to Polish Academy of Sciences, Warsaw, September 1976.

Software Reliability Predictions, with Dr. P. B. Moranda, MDAC Paper WD 2482, presented to the Federation for Automatic Control, Boston, and published in its proceedings, August 1975.

Can Statistics be Applied to Software - Historical Perspective, MDAC Paper WD 2531, presented to the Computer Science and Statistics 8th Annual Symposium on the Interface, Los Angeles, February 1975.

Applications of a Probability-Based Model to a Code-Reading Experiment, with Dr. P. B. Moranda, MDAC Paper WD 2067, presented to the Symposium on Software Reliability Sponsored by IEEE, New York, and published in its proceedings, April-May 1973.

Generalized Events-Oriented Simulation System (GESS) - A Performance Evaluation Tool, with Dr. G. S. Chung, MDAC Paper WD 2033, presented to Computer Performance Evaluation Users Group sponsored by National Bureau of Standards, Washington, D. C., published in proceedings, October 1972.

Software Reliability Research (with Dr. P. B. Moranda), MDAC Paper WD 1808, presented to the Conference on Statistical Methods for Evaluation of Computer Systems Performance, Providence, Rhode Island, and published in its proceedings, November 1971.

**PAUL B. MORANDA - TECHNICAL ADVISOR**

AB, Chemistry, Fresno State College, 1942.

MA, Mathematics, Ohio State University, 1948.

PhD, Mathematics, Ohio State University, 1953.

Dr. Moranda, now a Senior Information Systems Advisor at MDAC-HB, was the principal investigator of a contract with AFOSR to investigate the development of quantitative methods for software reliability. He is also working on software validation. During the first 9 months of his employment at MDAC he was principal investigator for software reliability IRAD. He developed mathematical models for software discrepancies and applied them to failure date to obtain estimates of the error content of software packages and estimates of their period of error-free performance. For 3 years subsequent to that assignment, he analyzed and developed logistics model for the YC-15 STOL aircraft. He is presently assigned to the Low Altitude Defense System.

Prior to joining MDAC in 1971, Dr. Moranda was Manager of Systems Analysis at Computer Real Time Systems, Newport Beach, was employed by North American Rockwell for 6 years, and was technical advisor to the director of data management systems at Autonetics Information Systems Division. He participated in several systems studies and development efforts. In the field of transportation, he was responsible for developing a system framework for the analysis of advanced marine transportation systems. Additionally, he participated in a quantitative analysis of the operations of the over-the-counter trading department of Merrill, Lynch, Pierce, Fenner, and Smith; an overview study of the American Stock Exchange; and a systems analysis of the U.S. Federal Court System.

In 1967 he was appointed scientific advisor to the director of management systems, in which capacity he developed methods of economic forecasting of sales and other business parameters employing random wavelet concepts. This work led to the formulation of a simulation model which predicted, in a balanced way, the sales, profit, cash flow, headcount, backlog, and facilities requirements.

On the California Integrated Transportation Study, he performed system synthesis, tradeoff studies, and mathematical analyses. In recent years, he presented five lectures in the transportation field at leading universities: October 1966, "Advanced Concepts in Transportation Planning," Carnegie Institute of Technology; June 1966 and June of 1967, "Application of Systems Analysis to Large Scale Systems - Transportation," University of California at Los Angeles; Spring 1967, organized and administered a full-time upper-division course in Dynamic Modeling for the University of California at Berkeley, in which he also delivered two lectures, including a summary of the California Transportation Study and the application of analytical techniques to the study of transportation problems.

Prior to joining Autonetics, Dr. Moranda was at the Aeronutronics Division of the Philco Ford Company, engaged in operations analysis. On a special assignment to the Ford Motor Company, he was responsible for mathematical modeling of the complete automobile production process. Other assignments included development of a methodology for handling fragmentary and unreliable data in a damage assessment center and development of a war game model for assessing military missile effectiveness. He held the position of Manager of Systems Analysis for 3 years.

Section 5  
INTERACTIONS

1. P. B. Moranda attended the 2nd Minnowbrook Workshop on Software Performance Evaluation, sponsored by Syracuse University and Rome Air Development Center. Participated in panel discussions on software modeling and metrics.
2. P. B. Moranda presented a paper, "Error Detection Models for Application During Program Development," to the National Computer Conference, Anaheim, California, on 22 May 1980.
3. P. B. Moranda presented the same paper at ACM's Pathways to System Integrity, Gaithersburg, Maryland, on 19 June 1980 (paper published in the Proceedings of the Conference).

Section 6  
REFERENCES

1. Metrics of Software Quality, MDAC Report MDC G7517, August 1979.
2. R. N. Motley and W. E. Brooks, "Statistical Prediction of Programming Errors," RADC-TR-77-175, Final Technical Report, Rome Air Development Center, AFSC, Griffiss Air Force Base, New York, May 1977.
3. Z. Jelinski, and P. B. Moranda, "Software Reliability Research" in Statistical Computer Performance Evaluation, Walter Freiburger, Ed., Academic Press, New York, 1972.
4. P. B. Moranda, "Estimation of a Priori Software Reliability," Computer Science and Statistics Interface Symposium, February 1975, Los Angeles, California.
5. T. A. Thayer, et. al, "Software Reliability Study," RADC-TR-76-238, Final Technical Report, AFSC, Griffiss Air Force Base, New York, August 1976.

Appendix A  
PTT OUTPUT

	Case 1	Case 2	Case 3	Summary
<b>MAIV</b>	66.67 Pc	33.33 Pc	33.33 Pc	100.00 Pc
1.	1	0	0	1
2.	1	1	0	2
3.	0	0	1	1
<b>TRIANG</b>	86.96 Pc	86.96 Pc	86.96 Pc	36.96 Pc
1.	1	1	1	3
2.	1	1	1	3
3.	0	0	0	0
4.	2	2	2	6
5.	1	1	1	3
6.	3	3	3	9
7.	2	2	2	6
8.	3	1	1	5
9.	1	1	1	3
10.	2	2	2	6
11.	1	1	1	3
12.	0	0	0	0
13.	2	2	2	6
14.	1	1	1	3
15.	2	2	2	6
16.	0	0	0	0
17.	1	1	1	3
18.	2	2	2	6
19.	3	3	3	9
20.	2	2	2	6
21.	3	3	3	9
22.	2	2	2	6
23.	3	3	3	9
<b>*Program</b>	84.67 Pc	80.77 Pc	80.77 Pc	88.16 Pc



# MAIN

## Segment Reference

1. (0-11)
2. (11,5-11)
3. (11-12)

- NO CDSMONITOR for MODULE "MAIN" -

# TRIANG

## Segment Reference

1. (0-3)
2. (3-1,34)
- \* 3. (34-37)
4. (37,2-3)
5. (37-38)
6. (34,36-37)
7. (3,5-8)
8. (3-11)
9. (11,7-8)
10. (11-13)
11. (13-16)
- \* 12. (13-10,34)
13. (13,20-22)
14. (22,20-22)
15. (22-27)
- \* 16. (27-21,32-33)
17. (33,23-27)
18. (33-31)
19. (27,20-31)
20. (31,20-31)
21. (31-33)
22. (13,15-18)
23. (8,10-11)

- NO CDSMONITOR for MODULE "TRIANG" -

# Trial Statistics

Number of trials(T)= 3

Value of Xi:

Xi	13 :	1
Xi	21 :	1

Number of Xi(N)= 2

	Case 4	Case 5	Case 6	Summary
MAIN	66.67 Pc	33.33 Pc	33.33 Pc	100.00 Pc
1.	1	0	0	1
2.	1	1	0	2
3.	0	0	1	1
TRIANG	87.61 Pc	86.96 Pc	36.96 Pc	86.96 Pc
1.	1	1	1	3
2.	1	1	1	3
3.	0	0	0	0
4.	2	2	2	6
5.	1	1	1	3
6.	3	3	3	9
7.	2	2	2	6
8.	2	2	1	5
9.	1	1	1	3
10.	2	2	2	6
11.	0	1	1	2
12.	0	0	0	0
13.	2	2	2	6
14.	1	1	1	3
15.	2	2	2	6
16.	0	0	0	0
17.	1	1	1	3
18.	2	2	2	6
19.	3	3	3	9
20.	2	2	2	6
21.	3	3	3	9
22.	2	2	2	6
23.	3	3	3	9
*Program	60.77 Pc	80.77 Pc	80.77 Pc	88.46 Pc

# MAIN

## Segment Reference

1. [0-11)
2. [11,5-11)
3. [11-12)

- NO CASSMONITOR for MODULE "MAIN" -

# TRIANG

## Segment Reference

1. [0-3)
2. [3-4,34)
- \* 3. [34-37)
4. [37,2-3)
5. [37-38)
6. [34,36-37)
7. [3,5-8)
8. [8-11)
9. [11,7-8)
10. [11-13)
11. [13-18)
- \* 12. [18-19,34)
13. [18,20-22)
14. [22,20-22)
15. [22-27)
- \* 16. [27-28,32-33)
17. [33,23-27)
18. [33-34)
19. [27,29-31)
20. [31,29-31)
21. [31-33)
22. [13,15-18)
23. [8,10-11)

- NO CASSMONITOR for MODULE "TRIANG" -

# Trial Statistics

Number of trials(T)= 6

Value of Xi:

XC	11 :	1
XC	21 :	1
XC	31 :	1

Number of Xi(N)= 3

	Case 7	Case 8	Case 9	Summary
<b>MAIN</b>	66.67 Pc	33.33 Pc	33.33 Pc	100.00 Pc
1.	1	0	0	1
2.	1	1	0	2
3.	0	0	1	1
<b>TRIANG</b>	86.96 Pc	78.26 Pc	86.96 Pc	96.96 Pc
1.	1	1	1	3
2.	1	1	1	3
3.	0	0	0	0
4.	2	2	2	6
5.	1	1	1	3
6.	3	3	3	9
7.	2	2	2	6
8.	1	0	3	4
9.	1	1	1	3
10.	2	2	2	6
11.	1	0	1	2
12.	0	0	0	0
13.	2	2	2	6
14.	1	1	1	3
15.	2	2	2	6
16.	0	0	0	0
17.	1	1	1	3
18.	2	2	2	6
19.	3	3	3	9
20.	2	2	2	6
21.	3	3	3	9
22.	2	2	2	6
23.	3	3	3	9
<b>*Program</b>	84.62 Pc	73.08 Pc	80.77 Pc	83.46 Pc

# MAIN

## Segment Reference

1. [0-11)
2. [11,5-11)
3. [11-12]

- NO C\$MONITOR for MODULE 'MAIN' -

# TRIANG

## Segment Reference

1. [0-3)
2. [3-4,34)
- \* 3. [34-37)
4. [37,2-3)
5. [37-38]
6. [34,36-37)
7. [3,5-3)
8. [8-11)
9. [11,7-3)
10. [11-13)
11. [13-15)
- \* 12. [18-19,34)
13. [18,20-22)
14. [22,20-22)
15. [22-27)
- \* 16. [27-28,32-33)
17. [33,23-27)
18. [33-34)
19. [27,29-31)
20. [31,29-31)
21. [31-33)
22. [13,15-19)
23. [8,10-11)

- NO C\$MONITOR for MODULE 'TRIANG' -

# Trial Statistics

Number of trials( $\tau$ )= 9

Value of  $X_i$ :

$X_i$	11 :	1
$X_i$	23 :	1
$X_i$	33 :	1
$X_i$	43 :	4

Number of  $X_i(N)= 1$



Appendix B  
OUTPUT FROM A CONSTRUCTED CASE  
AND  
CONSTRUCTED CASES LISTINGS

# OUTPUT FROM A CONSTRUCTED CASE

	Case	1	Summary
MAIN	56.57 Pc	66.67 Pc	
1.	1	1	
2.	0	0	
3.	1	1	
TRIANG	73.91 Pc	73.91 Pc	
1.	1	1	
2.	1	1	
3.	1	1	
4.	2	2	
5.	1	1	
6.	3	3	
7.	2	2	
8.	2	2	
9.	1	1	
10.	2	2	
11.	0	0	
12.	1	1	
13.	1	1	
14.	0	0	
15.	1	1	
16.	1	1	
17.	0	0	
18.	1	1	
19.	0	0	
20.	0	0	
21.	0	0	
22.	2	2	
23.	3	3	
*Program	73.98 Pc	73.03 Pc	

# MAIN

## Segment Reference

- 1. [0-10)
- 2. [10,5-10)
- 3. [10-11)

- NO C\$MONITOR for MODULE MAIN -

# TRANS

## Segment Reference

- 1. [0-3)
- 2. [3-4,36)
- 3. [36-39)
- 4. [39,2-3)
- 5. [39-40)
- 6. [35,38-39)
- 7. [3,5-3)
- 8. [8-11)
- 9. [11,7-3)
- 10. [11-13)
- 11. [13-1)
- 12. [19-20,36)
- 13. [19,21-23)
- 14. [23,21-23)
- 15. [23-2)
- 16. [29-30,34-35)
- 17. [35,24-29)
- 18. [35-35)
- 19. [29,31-33)
- 20. [33,31-33)
- 21. [33-35)
- 22. [13,15-19)
- 23. [3,10-11)

- Monitor Predicate -

	Predicate	Type	Minimum	Maximum
1.	5	Integer	1	
2.	CONT	Real	.0000000000000000E+00	.4000000010000000E+
3.	TIME	Real	.0000000000000000E+00	.0010000000000000E+

Trial Statistics

Number of trials(T)= 1

Value of  $Y_i$ :

Number of  $X_i(N)= 0$

# CONSTRUCTED CASES LISTINGS

```

PROGRAM MAIN
IMPLICIT INTEGER(1-7)

C
INTEGER IP(3)
REAL    A(4,3),R

C
DATA    A/0.0,4.0,3.0,5.0,2.0,4.3,0.0,7.1,9.0,0.0,11.0,12.0

C
0-      1      OPEN(UNIT=20,DEVICE='DSK:',FILE='BRAD.RES',ACCESS='SEQUENT')
C
2
3      N=3
4      NI=4
5      M=1
C
5      DO 10 K=1,M
6          WRITE(20,101)K,((A(I,J),J=1,N),I=1,N)
7          CALL TRIANG(IP,A,N)
8          WRITE(20,102)K,((A(I,J),J=1,N),I=1,N)
9-      10      CONTINUE
11      STOP
C
12      101    FORMAT(' BEFORE TRIANGULARIZATION (' ,I1,')'//3(3X,F20.10))
13      102    FORMAT(' AFTER  TRIANGULARIZATION (' ,I1,')'//3(3X,F20.10))
C
14      END

```

```

SUBROUTINE TRIANG(IP,A,N)
IMPLICIT INTEGER(A-Z)

```

C

```

INTEGER IP(3)
REAL A(4,3),T
REAL MONT ,TMON

```

C

```

0- 1      IP(N)=1
2      DO 5 K=1,N
3- 4      CSSMONITOR=INTEGER(K);
5          IF(K.EQ.N) GOTO 5
6          KP1=K+1
7          M=K
8- 9          DO 1 I=KP1,N
10- 11      IF(ABS(A(I,K)).GT.ABS(A(I,M))) =1
12          CONTINUE
13- 14      IP(K)=M
15          IF(M.NE.K) IP(N)=-IP(N)
16          T=A(M,K)
17          A(M,K)=A(K,K)
18          A(K,K)=T
19          MONT=ABS(T)
20      CSSMONITOR=REAL(MONT);
21      IF(T.EQ.0) GOTO 5
22- 23      DO 2 I=KP1,M
24          A(I,K)=-A(I,M)/T
25      DO 4 J=KP1,M
26          T=A(M,J)
27          A(M,J)=A(M,J)
28          A(K,J)=T
29          TMON=ABS(T)
30      CSSMONITOR=REAL(TMON);
31      IF(T.EQ.0) GOTO 4
32- 33      DO 3 I=KP1,M
34          A(I,J)=A(I,J)+A(I,M)*T
35      CONTINUE
36- 37      IF(A(K,K).EQ.0) IP(N)=0
38- 39      CONTINUE
40      RETURN
41      END

```

**DATE**  
**ILME**